Discrete Applied Mathematics 156 (2008) 3248–3266

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

The stack loading and unloading problem

Federico Malucelli [a], Stefano Pallottino [b,1], Daniele Pretolani [c,*]

[a] Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy

[b] Late of Dipartimento di Informatica, Università di Pisa, Italy

[c] Dipartimento di Scienze e Metodi dell'Ingegneria, Università di Modena e Reggio Emilia, via Amendola 2, I-42100 Reggio Emilia, Italy

ARTICLE INFO

Article history:
Received 14 October 2005
Received in revised form 27 April 2007
Accepted 21 May 2008
Available online 16 July 2008

Keywords:
Stowage problems
Dynamic programming
Computational complexity

ABSTRACT

When piling a set of items in a single stack, one often does not pay attention to the order. Real-life experience suggests that, whenever a specific item is suddenly requested, we need to dig very deep into the stack to extract it.

In this paper we investigate stack reordering strategies aiming at minimizing the number of *pop* and *push* operations. In particular we focus on three versions of the problem in which reordering can take place in different phases: when unloading the stack, when loading it or in both phases. We show that the first two variants can be solved in linear time, while for the third one we devise a dynamic programming method with quadratic complexity.

1. Introduction

Stacks are a very well known elementary data structures, largely used in many algorithmic frameworks and application fields. In this paper, rather than studying stacks as data structures or computational models, we focus on particular optimization problems arising when loading or unloading a stack. These problems have a direct practical application if we consider a stack from a physical point of view. For example, a train, a pallet truckload, a container pile in a ship can be modeled as stacks since loading and unloading operations can take place only in one extreme (the top or the rear) and usually elements can be popped or pushed one at a time, as in the case of pallets or containers.

In particular in freight train composition problems, the management of these operations is a key issue in the overall transportation optimization [5]. Consider a freight train traveling along a given route, and the operations that must be carried out at each station of the route. In each station, the train cars that arrived at their destination have to be detached from the train and new cars may possibly be added. This operation is called *classification* and is quite time consuming and also dependent on the capacity of the station yard. Therefore, one may aim at reordering the train cars in such a way that the time spent in adding and removing cars from the train is minimized.

Stack loading problems arise in relation with the optimization of storage yard operations in a container terminal, and in the so-called stowage planning of containerships. Containers are stored in stacks of limited height, and each container arriving at the terminal must be assigned an available stack. Here the goal is to optimize the *reshuffling*, that is, the reordering that becomes necessary whenever a container is assigned to a stack where some containers have earlier (estimated) retrieval time. The peculiarity of stowage planning problems is to deal with multiple stacks of limited height. Moreover, additional aspects must be taken into account such as the possibility of piling containers of different shapes, thus satisfying some rules, or the constraints deriving from the ship center of mass and the containers weight or other strength constraints. All

* Corresponding author. Tel.: +39 0522 522229; fax: +39 0522 522609.
E-mail address: daniele.pretolani@unimore.it (D. Pretolani).

[1] Stefano Pallottino unexpectedly passed by on April, 11 2004, when the theoretical developement of this work was already complete.

0166-218X/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.dam.2008.05.020

those aspects play usually a central role in the problem and cannot be ignored. Several works tackling these problems via mathematical programming approaches and heuristic algorithms have been proposed, see for example [3] for the stowage problem and [6,9] for a combined optimization of stowage and yard operations. A thorough review on the subject can be found in [8].

In [1,2] the stowage problem is tackled in a more abstract way, introducing stacks to model it. Some general problems, where some constraints are omitted, have been investigated proposing some algorithms for the single stack case and also trying to extend the approach to the multiple stack case.

Other problems may contain some stacking aspects, though stacking appears more as a constraint than an optimization issue. Consider the case of a vehicle (a truck, an AGV) that must be loaded and unloaded subject to LIFO (Last-In First-Out) constraints. Pick-up and delivery routing problems were addressed in [7,10]. This is the typical case where the vehicle can be represented by a stack. The LIFO constraints require that the goods to be unloaded and delivered are those on top of the stack. In this class of problems, however, the LIFO constraints influence the route definition, rather than vice-versa as in the stowage problems, where the route is fixed and the piled objects may be reshuffled.

In this paper we focus on the abstract version of the stacking problem with a single stack. A stack containing (at most) n elements is represented by an integer array S where $S[n]$ is the *bottom*, and the elements can be inserted and removed at the top, by means of *push* and *pop* operations. The elements have different *types* which are denoted by integers in the interval $[1, T]$. We assume that $T \leq n$, and that there exists at least one element of each type. Here we consider some optimization problems related with the loading and unloading operations that are performed to sort the elements in the stack.

In the *unloading* problem we have a full stack where the elements are not ordered. The stack must be emptied in T *stages* from 1 to T. During stage t all the elements of type t must be removed. This possibly requires to pop and push back some elements of type greater than t. The aim is to minimize the number of pushes.

In the *loading* problem we have an empty stack that must be loaded in $K < n$ *steps*. At each step a batch of elements has to be loaded, into the stack; possibly, some elements can be popped out of the stack, sorted, and pushed again into the stack. The goal is to obtain a completely ordered stack minimizing the number of pops.

In the *loading–unloading* problem we combine the two problems sketched above. We start with an empty stack, we load it in K steps and we unload it in T stages. The stack is not required to be ordered at the end of the loading phase. The aim is to minimize the sum of the pops in the loading phase plus the pushes in the unloading phase.

We assume that when out of the stack the elements can be sorted in any order, hence in particular they can be inserted into the stack in non-increasing order of type, which is the most suitable way for the problems we consider here. We also assume that the cost of sorting is negligible with respect to the cost of pop and push operations, as it usually happens in the applications that we mentioned above.

In this paper we give linear algorithms for the first and the second problem, and we propose an $O(n + KT)$ dynamic programming approach for the third one.

Even thought the setting that we describe is intentionally abstract and general, there are possible applications of these particular cases of stacking problems. For the loading–unloading problem one may think for example to a long haul transportation service (either road or maritime) where goods are first collected from a set of distributed clients then, after a long journey, they are delivered to another set of distributed clients [6]. Furthermore, in a road transportation service reshuffles may be impossible during the loading or the unloading phase, e.g. due to technical or schedule constraints; this gives rise to the unloading and loading problem, respectively.

Note that our loading–unloading problem is a particular case of the *One-Stack Overstowage Problem* (OSOP) presented in [1]. In that case, loading and unloading operations may occur at any moment, that is there is not a separation between the loading phase and the unloading one. The solution method proposed by Aslidis for OSOP solves the loading–unloading problem in cubic time $O(K + T)^3$. However, our loading and unloading problems do not fall into Aslidis model unless ad-hoc extensions are adopted, (see Section 5.2 in [1]). As a consequence, Aslidis' method may have a *worse* computational complexity when applied to our simpler problems. For example, the loading problem may take up to $O(T^3 + nT)$ time, where possibly $n \gg T^2$. Clearly, our specialized algorithms are much faster, being linear in n and at most quadratic in K and T.

We treat the unloading, loading and loading–unloading problems in Sections 2–4, respectively. Conclusions are reported in Section 5.

2. The stack unloading problem

We start with a full stack S of n elements where 1 is the top, i.e. the index of the first non empty element of S. The stack must be emptied in T stages from stage 1 to stage T. At stage t all the elements of type t must be removed, and this may require to pop some elements $t' > t$, that must be pushed back at the end of stage t. These elements can be pushed in any order, in particular in non increasing order. This operation is called a *reordering*. A reordering can be done at any stage t and can involve also elements in deeper positions with respect to the one strictly required, that is the position of the deepest element of type t. Here we consider the problem of unloading all the elements from the stack with the aim of minimizing the total number of pop and push operations. However, since the stack must be completely emptied, the number of pops is given by n plus the number of pushes. Therefore, our goal is to minimize the number of push operations.

An *unloading strategy* (*strategy* hereafter) is defined by the number of push operations performed at each stage t. For instance, a simple strategy can be obtained as follows: at stage t remove from the stack all elements down to the deepest

h	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$S[h]$	3	4	1	2	4	7	5	9	6	9	5	10	8	9	10	9

t	1	2	3	4	5	6	7	8	9	10
u_t	1	1	1	2	2	1	1	1	4	2
$U(t)$	1	2	3	5	7	8	9	10	14	16
$top(t)$	1	2	3	4	6	8	9	10	11	15
$deep(t)$	3	4	1	5	11	9	6	13	16	15

Fig. 1. Running example: a stack with 16 elements.

one of type t. However, in some stages it may be worthy to remove more elements than those strictly necessary, thus anticipating some work due in successive stages. In the extreme case, we may reorder all the elements in the first stage, thus obtaining a completely ordered stack, that can be emptied without further reorderings.

In the rest of this section, after introducing the basic notations, we show how to obtain an optimal unloading strategy. Our approach is as follows. First we devise a decomposition method, and solve the problem for *elementary substacks*, i.e. stacks that cannot be further decomposed. Then, we exploit this result to obtain a closed-form solution for the general case.

2.1. Notation and simplification

For a given stack S the number of elements of type t is denoted by u_t, while $U(t)$ is the number of elements of type at most t, that is:

$$U(t) = \sum_{j=1}^{t} u_j.$$

For notational convenience we let $u_0 = U(0) = 0$. The top of the stack at the beginning of stage $t > 1$, that is after we removed all elements of types at most $t-1$, is given by:

$$\text{top}(t) = U(t-1) + 1, \quad t = 2, \ldots, T, \tag{1}$$

while $\text{top}(1) = 1$. A most relevant information is the position of the deepest element of each type t in the initial configuration of the stack, denoted by:

$$\text{deep}(t) = \max\{h : S[h] = t, h = 1, \ldots, n\}, \quad t = 1, \ldots, T.$$

Finally, given a stack S of size n and two integers $1 \le h < k \le n$ we denote by $S[h, k] = \{S[h], S[h+1], \ldots, S[k]\}$ the *substack* of S containing the positions from h to k.

Example 1. Consider the stack S represented in Fig. 1, where $n = 16$ and $T = 10$; for each $t = 1, 2, \ldots, T$ the values u_t, $U(t)$, $\text{top}(t)$ and $\text{deep}(t)$ are reported.

In some situations, the stack unloading problem may be decomposed into subproblems. This occurs if S can be *split at* $\bar{h}$, that is, if we can identify a *separation index* $\bar{h} < n$ such that $S[h] \le t$ for $h = 1, \ldots, \bar{h}$ and $S[h] > t$ for $h = \bar{h}+1, \ldots, n$. Note that $\bar{h}$ is a separation index if and only if $\bar{h} = U(t)$ where $t = \max\{S[j] : j = 1, \ldots, \bar{h}\}$. In this situation, we can decompose the stack S into two substacks $S[1, \bar{h}]$ and $S[\bar{h}+1, n]$, that can be treated separately. In particular, this means that we can adopt a *decomposable* optimal strategy, that does not pop elements in $S[\bar{h}+1, n]$ during the first t stages. Without loss of generality, we can assume that a decomposable strategy is adopted whenever the stack can be split.

Note that a stack may have several separation indices, and thus may be split into several substacks. Moreover, a generic substack $S[\bar{k}+1, \bar{h}]$ may contain only elements of type t, and all of them: this happens if $S[h] < t$ for $h = 1, \ldots, \bar{k}$ and $S[h] > t$ for $h = \bar{h}+1, \ldots, n$. Clearly, in this situation the elements in the substack are unloaded at stage t without reordering, and we say that the substack is *ignored*.

Consider now a stack that cannot be split, and contains elements of different types. A further simplification occurs if the bottom part contains elements of type T, e.g., $S[i] = T$ for $n - k < i \le n$; note that we have $k < u_T$ here, otherwise, S could be split at $\bar{h} = n - k$. In this case we may consider a *reduced* stack, with $n - k$ elements, obtained by dropping the last k positions.

Example 1 (*Continued*). In the example of Fig. 1 a separation index is given by $\bar{h} = 5$; so, we can split the stack S into two substacks, S_1 of size 5 and S_2 of size 11, as shown in Fig. 2. Note that S_1 contains the elements of type at most 4. Note also that the bottom position in substack S_1 contains an element $S[5] = 4$, and thus can be dropped. Therefore, instead of S_1 we may consider a reduced substack $S_1^r = [3, 4, 1, 2]$, of size 4. □

We say that a stack is *elementary* if it cannot be split or reduced. Later on, we shall provide a solution method for elementary stacks. The general case (i.e., non-elementary stacks) can be dealt with adopting a decomposable strategy.

h	1	2	3	4	5
$S_1[h]$	3	4	1	2	4

h	6	7	8	9	10	11	12	13	14	15	16
$S_2[h]$	7	5	9	6	9	5	10	8	9	10	9

Fig. 2. Substacks of stack S.

h	6	7	8	9	10	11	12	13	14	15	16
$S([h])$			6	7	9	9	10	8	9	10	9

Fig. 3. The stack S_2 after stage 5.

i	0	1	2	3	4	5	6
IS_i	0	1	2	4	5	8	9
$deep(IS_i)$	0	3	4	5	11	13	16

Fig. 4. Increasing sequence of the stack in Fig. 1.

2.2. Increasing sequence

Here we define a particular subset of types (the *increasing sequence*) whose properties turn out to be crucial in our solution approach. The intuition behind this definition is captured by the following example.

Example 1 (*Continued*). Consider the stack S_2 in Fig. 2; we have deep(7) $<$ deep(6) $<$ deep(5). Suppose that in stage 5 we pop all the elements down to deep(5) $= 11$, obtaining the configuration given in Fig. 3. Note that types 6 and 7 can be removed without reordering in the corresponding stages. □

Consider two types t, t' with $t' < t$ and deep$(t') >$ deep(t). At the end of stage t' all the elements of type t have already been popped and pushed in at least one reordering operation. As a consequence (a formal proof is given later) at the beginning of stage t these elements appear on the top of S, and can be removed without any reordering. For this reason, we concentrate on those types t such that it does not exist any $t' < t$ with deep$(t') >$ deep(t), that is, those t satisfying:

$$S[h] > t, \quad \forall h : n \geq h > \text{deep}(t). \tag{2}$$

By definition, type 1 and type $S[n]$ satisfy (2). Moreover, if a pair t, t' satisfy (2) and $t' < t$ then deep$(t') <$ deep(t). Thus the m types satisfying (2) define an *increasing sequence* $IS_1, \ldots, IS_m$, $1 \leq m \leq T$, having the following properties:

- $IS_i < IS_{i+1}$, for each $1 \leq i < m$;
- deep$(IS_i) <$ deep(IS_{i+1}), for each $1 \leq i < m$;
- $IS_1 = 1$, and $IS_m = S[n]$.

For notational convenience, we let $IS_0 = 0$ and deep$(0) = 0$.

Example 1 (*Continued*). The increasing sequence for the stack in Fig. 1 is shown in Fig. 4; here, $m = 6$. □

We can now prove the property claimed above.

Property 1. *Let $IS_i = t$ and $IS_{i+1} = t'$ for some $i = 1, \ldots, m-1$; the following claims hold true:*

1. *the stack elements in positions between* deep(t) *and* deep(t') *have types at least t', i.e., $S[h] \geq t'$ for* deep$(t) < h \leq$ deep(t')*;*
2. *at the end of stage t, all the elements of type $t+1, \ldots, t'-1$ (if any, i.e., if $t' > t+1$) appear ordered on top of the stack.*

Proof. To prove the first claim, let $\hat{h}$ be the deepest position containing an element of type smaller than t'. It is easy to see that type $\hat{t} = S[\hat{h}]$ satisfies (2). Since clearly $\hat{h} <$ deep(t'), if $\hat{h} >$ deep(t) we have $t < \hat{t} < t'$, contradicting the assumption $IS_i = t$ and $IS_{i+1} = t'$.

The second claim then follows from the first claim and the definition of IS. Indeed, we have $S[h] > t'$ for all $h >$ deep(t), thus all the elements of type smaller than t' have been reordered at the end of stage t. □

In light of Property 1, the sequence IS can be computed in $O(n)$ time by processing the stack from the bottom to the top, as described below. Since the elements in IS are found in reverse order, we need the auxiliary array RIS to store the reverse IS sequence.

Step 0 let $RIS[1] = S[n]$, $i = 1$ and $h = n - 1$;
Step 1 if $S[h] < RIS[i]$ then let $i = i + 1$ and $RIS[i] = S[h]$;
Step 2 if $S[h] = 1$ then go to Step 3; otherwise, let $h = h - 1$ and go to Step 1;
Step 3 let $m = i$; while $i > 0$ do $IS_{m+1-i} = RIS[i]$, $i = i - 1$; STOP.

The increasing sequence IS provides a method for decomposing the loading problem efficiently. Suppose that $\bar{h}$ is a separation index, and let $t = S[\bar{h}]$; clearly, type t satisfies condition (2), and thus belongs to IS. Moreover, given type $t = IS_j$ for some $j = 1, \ldots, m-1$, $\bar{h} = \text{deep}(t)$ is a separation index if and only if

$$\bar{h} = U(IS_{j+1} - 1). \tag{3}$$

Condition (3) follows since IS_{j+1} is the smallest type in substack $S[\bar{h}+1, n]$, thus $IS_{j+1} - 1$ is the largest type in $S[1, \bar{h}]$. Now consider a substack $S[\bar{k}+1, \bar{h}]$ that cannot be split; here, $\bar{k} = \text{deep}(IS_i)$, where possibly $i = 0$, and $\bar{h} = \text{deep}(IS_j)$. Assume the following condition holds:

$$IS_j = \max_{\bar{k}<h\leq\bar{h}} S[h]. \tag{4}$$

Note that condition (4) holds if either $j = m$ (i.e., $\bar{h} = n$) and $S[n] = T$, or if $IS_j = IS_{j+1} - 1$; both cases can be checked in constant time. In this situation, two cases are possible. If $i = j - 1$ we conclude that $S[\bar{k}+1, \bar{h}]$ contains only elements of type IS_j, and can be ignored. Otherwise, we reduce $S[\bar{k}+1, \bar{h}]$ by dropping elements of type IS_j, obtaining an elementary substack $S[\bar{k}+1, \text{deep}(IS_{j-1})]$. In both cases, $\text{deep}(IS_{j-1})$ is the deepest position containing an element smaller than IS_j, thus the number of elements that are dropped or ignored is given by

$$\bar{h} - \text{deep}(IS_{j-1}) = U(IS_j) - \text{deep}(IS_{j-1}). \tag{5}$$

In conclusion, we can decompose a stack unloading problem into subproblems on elementary stacks in $O(m)$ time. Moreover, the decomposition of S induces an obvious decomposition of IS, since a substack $S[\text{deep}(IS_i)+1, \text{deep}(IS_j)]$ contains the subsequence $\{IS_{i+1}, \ldots, IS_j\}$ of IS. Note that this decomposition excludes each IS_j that satisfies conditions (3) and (4).

Example 1 (*Continued*). Consider the increasing sequence for the stack in Fig. 1. Eq. (3) is satisfied only for $j = 3$; we have $\text{deep}(IS_3) = \text{deep}(4) = 5$ and $U(IS_{j+1} - 1) = U(5 - 1) = 5$. Indeed, $\bar{h} = 5$ is a separation index in the stack, see Fig. 2. Furthermore, since $IS_{j+1} - 1 = IS_j$ for $j = 3$, Eq. (4) is satisfied. The substack $S_1 = S[1, 5] = S[\text{deep}(IS_0)+1, \text{deep}(IS_3)]$ cannot be ignored, since $0 \neq 3 - 1$. According to Eq. (5), we can drop one element (namely $S[\bar{h}]$) since for $j = 3$ we have $U(IS_j) = U(4) = 5$ and $\text{deep}(IS_{j-1}) = \text{deep}(IS_2) = \text{deep}(2) = 4$. Thus we obtain the reduced substack $S_1^r = [3, 4, 1, 2]$ mentioned above. □

2.3. Optimal strategies, and closed form solution

In this section we devise a method for finding an *optimal* strategy, i.e. a strategy requiring a minimum number of push operations. Recall that, in light of Property 1, it is not necessary to perform any reordering at a stage t such that $IS_i < t < IS_{i+1}$ for some $1 \leq i < m$. It is easy to see that it is not worth doing a reordering at such stage t, since the same effect may be obtained, with less push operations, at stage $t + 1$ or later, i.e. after unloading t. This fact has a relevant consequence:

Proposition 1. *There always exist optimal strategies where push operations are performed at stages in the increasing sequence only.*

Note however that reordering at *each* stage in IS is not compulsory. Indeed, during a stage IS_i we may perform some operations that are due at later stages, so that reordering is no longer needed at those stages. Consider a strategy that performs a reordering at stage $t = IS_i$, followed by a reordering at stage $t' = IS_{j+1}$, with $j > i$. We call this situation an *anticipated reordering of* IS_j *at* IS_i. In some cases, anticipated reordering can reduce the overall number of push operations, as shown by the following example.

Example 1 (*Continued*). Consider the reduced substack $S_1^r = [3, 4, 1, 2]$, where we have $IS = \{1, 2\}$. If we reorder at stages $IS_1 = 1$ and $IS_2 = 2$ we perform four push operations, since two elements (of type 3 and 4) are pushed back at each stage. However, in stage $IS_1 = 1$ we can anticipate the reordering of IS_2, i.e., pop all the elements down to $\text{deep}(2) = 4$. This requires three push operations, but gives the stack $S' = [2, 3, 4]$, that can be unloaded without further reordering. □

In fact, a strategy is completely described by specifying the set of stages where the reordering is anticipated, or equivalently, by specifying the stages in IS where a reordering is performed. In order to find an optimal strategy, we need to evaluate the number of push operations required by an anticipated reordering, which allows us to detect the cases when we can improve on the simple strategy where no reordering is anticipated.

Let us first concentrate on elementary stacks. First of all, observe that a reordering is always necessary at stage $IS_1 = 1$, otherwise deep(1) would be a separation index. The case where IS has length $m = 1$ is trivial, since the stack is completely ordered at the end of stage 1. In the following we assume $m \geq 2$.

Lemma 1. *Consider an anticipated reordering of* IS_j *at stage* IS_i*, where* $1 \leq i < j \leq n$*. The number of push operation performed at stage* IS_i *is given by:*

$$\pi(i, j) = \text{deep}(IS_j) - \text{top}(IS_i + 1) + 1 = \text{deep}(IS_j) - U(IS_i). \tag{6}$$

Proof. At the end of stage IS_i all the elements of type at most IS_j must appear ordered on top of the stack. By Property 1 these elements are contained in $S[\text{top}(IS_i), \text{deep}(IS_j)]$, so we do not need to pop elements below $\text{deep}(IS_j)$. Moreover, $S[\text{top}(IS_i), \text{deep}(IS_j)]$ contains some elements of type greater than IS_j, otherwise $\text{deep}(IS_j)$ is a separation index, contradicting the hypothesis that the stack is elementary. Therefore, we need to pop all the elements of type at most IS_j, that is, we must pop down to position $\text{deep}(IS_j)$, which leaves $\text{deep}(IS_j) - U(IS_i)$ elements to push. □

The number of push operations needed at stage IS_i, with no anticipated reorderings, is denoted by $\pi(i, i) = \text{deep}(IS_i) - U(IS_i)$. Now consider an anticipated reordering of type IS_l at stage IS_i, and compare to the case where we reorder at IS_i and at IS_j, for some $i < j \leq l$. More precisely, in the latter case we reorder IS_{j-1} at stage i, where possibly $i = j - 1$, and reorder IS_l at stage IS_j, where possibly $j = l$; the number of push operations is $\pi(i, j-1) + \pi(j, l)$. The difference is given by:

$$\begin{aligned}\pi(i, j-1) + \pi(j, l) - \pi(i, l) &= \text{deep}(IS_{j-1}) - U(IS_i) + \text{deep}(IS_l) - U(IS_j) - (\text{deep}(IS_l) - U(IS_i)) \\ &= \text{deep}(IS_{j-1}) - U(IS_j) = \delta_j.\end{aligned}$$

In this situation, the further reordering at stage IS_j is worth doing if and only if the value δ_j is negative. Note that the value δ_j is independent from IS_i and IS_l, but is expressed only in terms of the intermediate stage IS_j. This means that the choice of performing a reordering in a stage IS_j does not depend on the choices in earlier or later stages. Based on this observation, we can define:

$$\delta_j = \text{deep}(IS_{j-1}) - U(IS_j), \quad j = 1, \ldots, m; \tag{7}$$

and prove the following result.

Theorem 1. *We obtain an optimal strategy for an elementary stack by reordering at each step IS_j such that $\delta_j < 0$; the minimum number of push operations is:*

$$U^* = n + \sum_{j=1}^{m} \min\{0, \delta_j\}. \tag{8}$$

Proof. The claims are a consequence of the following property: a strategy reordering at stages in the set $I \subseteq IS$ requires $U^{(I)}$ push operations, where

$$U^{(I)} = n + \sum_{IS_j \in I} \delta_j.$$

This property can be proved by induction on the length k of I. Recall that a reordering at stage 1 is always necessary. For $k = 1$ we have $I = \{IS_1\}$, i.e., the simple strategy that reorders the whole stack at the first stage, taking $n - U(1) = n + \delta_1$ pops. The induction step is a direct application of the definition of the values δ. □

Observe that the formula (8) holds true also for the trivial case where $m = 1$. Since the increasing sequence can be computed in $O(n)$ time, we conclude that the unloading problem for elementary stacks can be solved in linear time.

As discussed before, once we have a solution method for elementary stacks, we can solve the general case by adopting a decomposition approach. Since we can find all the elementary substacks in time $O(m)$, and solve each substack in linear time with respect to its length, we obtain an $O(n)$ algorithm for the general case. In the following, we discuss the extension of Theorem 1 to non-elementary stacks. In particular, we show that formula (8) holds true in general. This result will be exploited in Section 4 to solve the Loading–Unloading problem.

Suppose that the stack S is decomposed into q elementary substacks $S^{(s)} = S[k^{(s)}, h^{(s)}]$, $s = 1, \ldots, q$, as discussed earlier. Let $n^{(s)}$ be the length of $S^{(s)}$, and let $IS^{(s)} = \{i^{(s)}, \ldots, m^{(s)}\}$ denote the subsequence of IS contained in substack $S^{(s)}$. Moreover, let

$$R = \{IS_1, \ldots, IS_m\} \setminus \bigcup_{s=1}^{q} IS^{(s)}$$

the (possibly empty) set of types in IS that do not belong to any subsequence. As discussed before, we have $IS_j \in R$ if and only if IS_j satisfies conditions (3) and (4), that is, if we ignored or dropped k elements of type IS_j during the decomposition process. According to (5), in this situation we have:

$$k = U(IS_j) - \text{deep}(IS_{j-1}) = -\delta_j$$

and we conclude that:

$$n = \sum_{s=1}^{q} n^{(s)} - \sum_{IS_j \in R} \delta_j.$$

Recall that a decomposable strategy deals with each elementary substack $S^{(s)}$ separately. As long as we have $i, j \in IS^{(s)}$, Lemma 1 holds true, in particular, the anticipated reordering of j at i requires $\pi(i, j)$ operations. We conclude that each substack $S^{(s)}$ requires $U^{(s)}$ push operations, where

$$U^{(s)} = \pi(i^{(s)}, m^{(s)}) + \sum_{j=i^{(s)}+1}^{m^{(s)}} \min\{\delta_j, 0\} = n^{(s)} + \sum_{j=i^{(s)}}^{m^{(s)}} \min\{\delta_j, 0\}$$

is obtained from (8). The total number of push operations is given by:

$$\begin{aligned}
\sum_{s=1}^{q} U^{(s)} &= \sum_{s=1}^{q} \left(n^{(s)} + \sum_{j=i^{(s)}}^{m^{(s)}} \min\{\delta_j, 0\} \right) \\
&= \sum_{s=1}^{q} n^{(s)} + \sum_{s=1}^{q} \sum_{j=i^{(s)}}^{m^{(s)}} \min\{\delta_j, 0\} \\
&= n + \sum_{IS_j \in R} \delta_j + \sum_{s=1}^{q} \sum_{j=i^{(s)}}^{m^{(s)}} \min\{\delta_j, 0\} \\
&= n + \sum_{j=1}^{m} \min\{\delta_j, 0\}.
\end{aligned}$$

We summarize the above discussion in the following theorem.

Theorem 2. *The stack unloading problem can be solved in $O(n)$ time. We obtain an optimal unloading strategy by reordering at each step IS_j that does not satisfy (4) and such that $\delta_j < 0$; the minimum number of push operations is provided by (8).*

Example 1 (*Continued*). The sequence $\{\delta_1, \ldots, \delta_6\}$ for the stack S is $\{-1, 1, -1, -2, 1, -1\}$, thus $U^* = 16 - 5 = 11$. We perform a reordering at stages $IS_1 = 1$, $IS_4 = 5$ and $IS_6 = 9$; we do not reorder at stage $IS_3 = 4$, since IS_3 satisfies (4). As discussed earlier, we perform three pushes at stage 1; at stage 5 we push six elements, namely $\{6, 7, 8, 9, 9, 10\}$, while at stage 9 we push two elements $\{10, 10\}$. □

3. The stack loading problem

In this case, starting with an empty stack, a *loading sequence* $B = \{B(1), \ldots, B(K)\}$ of *batches* of elements must be loaded in K successive *steps*. The batch $B(s)$ is a non-empty set of elements to be loaded in step s. At the beginning of step s, some elements can be popped from the stack, and then pushed back together with the elements of $B(s)$. Elements can be pushed into the stack in any order, in particular, in non-increasing order of type. At the end of the loading phase, we must obtain an ordered stack, that can be unloaded without performing any push. The goal is to minimize the total number of pop operations, since the total number of pushes is given by n plus the number of pops.

Also in this case, we refer to the sequence of pop and push operations made during a step as a reordering. The *depth* of a reordering is the deepest position (maximum stack index) where an element is popped. A *loading strategy* is defined by the number of pops performed at each step, or equivalently, by the steps where a reordering is performed, together with the corresponding depths. Two simple strategies can be pointed out: making only one reordering at the final step, or performing a reordering at each step to maintain an ordered partial stack throughout the loading phase.

Since our goal is to minimize the total number of pops, we can restrict ourselves to considering one particular kind of loading strategies.

Definition 1. A loading strategy is *dominating* when the depth of each reordering is smaller than the depth of any previous reordering.

Indeed, if a reordering pops elements in deeper positions with respect to a previous reordering, one could obtain the same stack configuration skipping the first reordering, thus saving some pops. Note that the strategy performing a single reordering at step K is dominating, which is not necessarily the case for the strategy that applies a reordering at each step.

3.1. Notation and decomposition

For each batch $B(s)$ let high(s) denote the maximum type contained in $B(s)$. The number of elements in the stack after step s is denoted by:

$$L(s) = \sum_{j=1}^{s} |B(s)|$$

s	1	2	3	4	5	6	7	8
$B(s)$	$\{8,10\}$	$\{6,8,8\}$	$\{9\}$	$\{7,9\}$	$\{5,6\}$	$\{4\}$	$\{3,5\}$	$\{1,2,4\}$
$high(s)$	10	8	9	9	6	4	5	4
$L(s)$	2	5	6	8	10	11	13	16

Fig. 5. A loading sequence B.

s	1	2	3	4
$B^{(1)}(s)$	$\{8\}$	$\{6,8,8\}$	$\{9\}$	$\{7,9\}$
$high^{(1)}(s)$	8	8	9	9
$L^{(1)}(s)$	1	4	5	7

s	6	7	8
$B^{(2)}(s)$	$\{4\}$	$\{3,5\}$	$\{1,2,4\}$
$high^{(2)}(s)$	4	5	4
$L^{(2)}(s)$	1	3	6

Fig. 6. Elementary loading sub-sequences $B^{(1)}$ and $B^{(2)}$.

where $L(K) = n$ is the *size* of the loading sequence, that is, the size of the stack at the end of the loading phase. For notational convenience, we introduce step $s = 0$ to denote the state before the loading phase, and define $L(0) = 0$ accordingly.

Example 2. Consider the sequence B represented in Fig. 5, where $n = 16$, $T = 10$, and $K = 8$; for each step $s = 1, 2, \ldots, K$ the values $L(s)$ and high(s) are reported.

Possibly, the loading sequence allows to decompose or simplify the loading problem. Suppose that, for type t and step $s < K$, the elements in the batches $B(1), \ldots, B(s)$ have types greater than or equal to t, while the elements in $B(s+1), \ldots, B(K)$ have types smaller than or equal to t. In this case, we can decompose the loading problem into two subproblems by *splitting* the sequence B at s. In the first subproblem, we must load the batches $B(1), \ldots, B(s)$ in the deepest $L(s)$ positions of the stack. In the second subproblem, we must load the batches $B(s+1), \ldots, B(K)$ in the remaining $n - L(s)$ positions. This corresponds to adopting a decomposable loading strategy, where we do not pop any element in the deepest $L(s)$ positions of the stack during steps $s+1, \ldots, K$. We can assume that a decomposable strategy is adopted whenever the loading sequence can be split. In general, B may be split into several subsequences. Note that if B can be split both at s and at $s-1$ we obtain a subsequence containing the single batch $B(s)$. In this situation, the elements in $B(s)$ are directly loaded in their final positions at step s, and we say that batch $B(s)$ is *ignored*.

Consider now a loading sequence B that cannot be split, and contains more than one batch. Suppose that the first batch contains $b > 0$ elements of type T: these elements can be ignored, since they are loaded at the bottom of the stack in step 1, and do not need to be popped in the remaining steps. In this case, we can simply consider a loading sequence of size $n - b$, dropping elements of type T from $B(1)$. Moreover, if $B(1)$ contains *all* the elements of type T, we obtain a sequence with $T - 1$ types instead of T; in this case, we can drop from $B(1)$ the elements of type $T - 1$, and so on. Note however that we cannot drop all the elements in $B(1)$, since otherwise we could split B at $s = 1$, which contradicts our assumption.

We say that a loading sequence is *elementary* if it cannot be split, and high$(1) < T$. In the following we shall show how to solve the loading problem for an elementary sequence, and how to decompose a sequence B into elementary subsequences, possibly identifying batches that can be simplified or ignored. Clearly, this allows us to solve the loading problem for arbitrary sequences.

Example 2 (*Continued*). In the example of Fig. 5, we can split the sequence B at step 4 and at step 5. Therefore, we can ignore batch $B(5)$, and consider two subsequences $B^{(1)} = \{B(1), \ldots, B(4)\}$ and $B^{(2)} = \{B(6), \ldots, B(8)\}$. Moreover, we can drop from $B(1)$ the element of type 10. In Fig. 6 we show the two resulting elementary sub-sequences, namely, $B^{(1)} = \{B(1), \ldots, B(4)\}$, with types $6, \ldots, 9$ and size $n^{(1)} = 7$; and $B^{(2)} = \{B(6), \ldots, B(8)\}$, with types $1, \ldots, 5$ and size $n^{(2)} = 6$. Values high and L are modified accordingly. □

3.2. Stable elements and non-increasing sequence

During the loading process, at the end of an intermediate step, the deepest portion of the stack may appear ordered as in the final configuration, while the elements in the remaining portion need to be ordered in the next steps.

Definition 2. At a given step, the elements in the stack that will not be popped in any subsequent step are called *stable elements*; the corresponding positions are called *stable positions*.

The goal of a reordering is to push some elements in stable positions, that is, to expand the stable portion of the stack. Therefore, a reordering requires to pop all the non-stable elements, thus its depth is exactly $n - k$, where k is the number of stable positions. Clearly, it is not sensible to perform a reordering at step s if this does not increase the number of stable elements. Observe that an element of type t cannot become stable in step s if a batch $B(s')$ with $s' > s$ contains an element of type $t' > t$. Therefore, a batch $B(s)$ contains some elements that can become stable at step s only if the following condition is satisfied:

$$\text{high}(s) \geq \text{high}(s') \quad \forall s' : K \geq s' > s. \tag{9}$$

Table 1
Stable elements at steps in BI

i	NS_i	ST_i	*Stable elements*	*Non-stable elements*
1	10	1	{10}	{8}
2	9	2	{10, 9}	{8, 8, 8, 6}
3	9	8	{10, 9, 9, 8, 8, 8, 7, 6}	{}
4	6	10	{10, 9, 9, 8, 8, 8, 7, 6, 6, 5}	{}
5	5	12	{10, 9, 9, 8, 8, 8, 7, 6, 6, 5, 5, 4}	{3}

Note that condition (9) is satisfied at step $s = K$. The steps satisfying condition (9) define a sub-sequence BI of B, where $BI = \{BI_1, \ldots, BI_m\}$, $1 \le m \le K$, has the following properties:

- $BI_i < BI_{i+1}$, for each $1 \le i < m$;
- $\text{high}(i) \ge \text{high}(i+1)$, for each $1 \le i < m$;
- $BI_m = K$;

The corresponding maximum types form a non-increasing sequence of values $NS = \{NS_1, \ldots, NS_m\}$, where $NS_i = \text{high}(BI_i)$ for each $1 \le i \le m$. We refer to NS, and by extension also to BI, as the *non-increasing sequence*, since it may happen that $NS_i = NS_{i+1}$ for some $1 \le i < m$. For notational convenience, we define $BI_0 = 0$.

Example 2 (*Continued*). In the example of Fig. 5 the non-increasing sequence has length $m = 6$, and is described by $BI = \{1, 3, 4, 5, 7, 8\}$ and $NS = \{10, 9, 9, 6, 5, 4\}$, as shown in the following table. □

s	1	2	3	4	5	6	7	8
$B(s)$	{8, 10}	{6, 8, 8}	{9}	{7, 9}	{5, 6}	{4}	{3, 5}	{1, 2, 4}
i, NS_i	1, 10		2, 9	3, 9	4, 6		5, 5	6, 4

□

Property 2. *Consider two steps BI_i and BI_{i+1}, for $1 \le i < m$. For each batch $B(s)$ with $BI_i < s < BI_{i+1}$ (if any, i.e., if $BI_i + 1 < BI_{i+1}$) we have* $\text{high}(s) < NS_{i+1}$. *Moreover, we have:*

$$NS_i \ge NS_{i+1} = \max_{s > BI_i} \text{high}(s).$$

Proof. To prove the first claim, let $\hat{t} = \text{high}(\hat{s})$ be the largest type in batches $B(BI_i + 1), \ldots, B(BI_{i+1} - 1)$, that is:

$$\hat{s} = \arg \max_{BI_i < s < BI_{i+1}} \text{high}(s).$$

If $\hat{t} \ge NS_{i+1}$ then $\hat{s}$ satisfies Condition (9), contradicting the assumption $\hat{t} \notin BI$. The second claim then follows from the first one and the definition of non-increasing sequence. □

Exploiting Property 2 we can find the sequence BI efficiently, by processing the batches in reverse order as described below. Here we use the array RBI to store the reverse sequence BI.

Step 0 let $RBI[1] = K$, $T_{\min} = \text{high}(K)$, $i = 1$ and $s = K - 1$;
Step 1 if $\text{high}(s) \ge T_{\min}$ then let $i = i + 1$, $RBI[i] = s$ and $T_{\min} = \text{high}(s)$;
Step 2 if $s = 1$ then go to Step 3; otherwise, let $s = s - 1$ and go to Step 1.
Step 3 let $m = i$; while $i > 0$ do $BI_{m+1-i} = RBI[i]$, $i = i - 1$; STOP.

Assuming that the values $\text{high}(s)$ are given, building the non-increasing sequence requires $O(K)$ time.

Let us denote by ST_i, $i = 1, \ldots, m$, the maximum number of elements that can become stable at the end of step BI_i. These elements must have types greater than or equal to the maximum type in batches $BI_i + 1, \ldots, K$, that is, greater than or equal to NS_{i+1}. Denoting by $C(s, t)$ the number of elements of type t contained in batch $B(s)$, we have:

$$ST_i = \sum_{s=1}^{BI_i} \sum_{t=NS[i+1]}^{T} C(s, t), \quad 1 \le i < m.$$

Clearly, we have $ST_m = n$; we also define $ST_0 = 0$. The values ST can be computed in overall $O(n)$ time, as shown in the Appendix.

Example 2 (*Continued*). In the example of Fig. 5 we have the sequence $ST = \{1, 2, 8, 10, 12, 16\}$; stable elements at BI_i, $1 \le i \le 5$, are shown in Table 1. Note that we have $ST_i = L(BI_i)$ (that is, all the elements are stable) for $i = 3, 4$, and the loading sequence B can be split at $BI_3 = 4$ and $BI_4 = 5$. □

The sequences BI, NS and ST can be used to decompose the loading problem efficiently. Indeed, B can be split at step s if and only if all the elements in batches $B(1), \ldots, B(s)$ can become stable at step s. In turn, this implies that s belongs to BI, that is, $s = BI_i$ for some $1 \le i < m$, and $ST_i = L(BI_i)$. This provides us with a simple method for decomposing the loading

sequence. We can also easily identify a batch $B(s)$ that can be ignored: in this case, B can be split at s and $s-1$, thus we have $s = BI_i = BI_{i-1} + 1$ for some $1 \le i < m$.

Now consider a generic subsequence $B' = \{B(s), \ldots, B(s')\}$ that cannot be split and such $s = BI_i + 1 < s' = BI_j$. Note that B' induces a subsequence $BI' = \{BI_{i+1}, \ldots, BI_j\}$ of BI. We can easily check if $B(s)$ contains some elements that can be dropped: since these elements can become stable at step s, we have $s = BI_i + 1 = BI_{i+1}$. If this is the case, we remove $ST_{i+1} - ST_i$ elements from $B(s)$, obtaining a reduced batch $B''(s)$ and an elementary subsequence $B'' = \{B''(s), \ldots, B(s')\}$. Note that B'' induces the subsequence $BI'' = BI_{i+2}, \ldots, BI_j$ of BI, obtained from BI' by dropping $s = BI_{i+1}$.

According to the above observations, we can decompose a loading sequence into elementary loading sub-sequences in time $O(K)$, once the values BI, NS and ST are given. Accordingly, we obtain a decomposition of BI into subsequences; note that these subsequences do not contain step $s = BI_i$ if batch $B(s)$ can be ignored or simplified, in which case k elements are ignored or dropped, where:

$$k = ST_i - ST_{i-1} = ST_i - L(BI_{i-1}). \tag{10}$$

Example 2 (*Continued*). Consider the example in Fig. 5, where BI can be split at $BI_3 = 4$ and $BI_4 = 5$; for subsequence $\{B(1), \ldots, B(4)\}$ we obtain the non-increasing (sub)-sequence $\{BI_1, BI_2, BI_3\} = \{1, 3, 4\}$, while for $\{B(6), \ldots, B(8)\}$ we obtain $\{BI_5, BI_6\} = \{7, 8\}$. Here we have $BI_1 = 1$ and $ST_1 = 1$, indeed, we can ignore the element of type 10 in batch $B(1)$. As a result, we obtain the elementary sub-sequence $B^{(1)}$ shown in Fig. 6; the corresponding non-decreasing sequence $BI^{(1)} = \{BI_2, BI_3\} = \{3, 4\}$ is obtained after removing BI_1. □

3.3. Optimal strategies, and closed form solution

We now devise a solution method finding an optimal loading strategy. Let us first consider the steps that do not belong to the sequence BI. According to Property 2, for each $1 \le i \le m$ and for each step s such that $BI_{i-1} < s < BI_i$, the elements in batch $B(s)$ cannot become stable before step BI_i. Therefore, it is not worth to reorder at such step s, since the same stable elements may be obtained, with less pop operations, by reordering at step BI_{i-1}. This allows us to concentrate on optimal strategies where reorderings are performed at steps in BI. However, the reordering at step BI_i may be postponed to a later step BI_j, with $j > i$. We call this choice a *delayed reordering* of batches $B(BI_i), \ldots, B(BI_{j-1})$. In order to find an optimal loading strategy we have to choose the steps in BI where we perform a reordering, or equivalently, we have to choose those steps where reordering is delayed.

Example 2 (*Continued*). The loading sub-sequence $B^{(1)} = \{B(1), \ldots, B(4)\}$ in Fig. 6 provides an example where delayed reordering is worth doing. The non-increasing sequence for $B^{(1)}$ is $BI^{(1)} = \{3, 4\}$. At step $BI_1^{(1)} = 3$ the stack configuration is [6, 8, 8, 8]; a reordering requires four pops, and yields the stack [6, 8, 8, 8, 9]; reordering at step $B_2^{(1)} = 4$ requires four more pops. If we do not reorder at step 3, at the beginning of step 4 we have the stack [9, 6, 8, 8, 8], which requires only five pops. □

Let us now concentrate on solving the loading problem for an elementary loading sequence B. Recall that we have $BI(1) > 1$ and $ST_i < L(BI_i)$ for each $1 \le i < m$. Therefore, a reordering is always necessary at step $BI_m = K$. To avoid trivial cases, we assume $m > 1$.

Let us suppose that at the end of a given step BI_i the stack contains ST_i stable elements. Note that this happens if either $i = BI_i = ST_i = 0$, or if we perform a reordering at step BI_i. Let us evaluate the effort of making next reordering at step BI_j, $j > i$. At the beginning of step BI_j, the stack contains $L(BI_j - 1)$ elements, out of which ST_i are stable; the reordering requires to pop all the $\rho(i, j)$ non-stable elements, where:

$$\rho(i, j) = L(BI_j - 1) - ST_i. \tag{11}$$

Now, let us compare the number of pops needed to make a reordering at steps BI_j and BI_k, $i < j < k \le m$, with the number of pops in case we delay reordering operations until step BI_k; in other words, we have to compare $\rho(i, j) + \rho(j, k)$ with $\rho(i, k)$. The difference is:

$$\begin{aligned}\rho(i, j) + \rho(j, k) - \rho(i, k) &= L(BI_j - 1) - ST_i + L(BI_k - 1) - ST_j - L(BI_k - 1) + ST_i \\ &= L(BI_j - 1) - ST_j.\end{aligned}$$

It should be noted that this difference is independent from BI_i and BI_k, and is expressed only in terms of the intermediate step BI_j. Therefore, we can define this difference as:

$$\theta_j = L(BI_j - 1) - ST_j, \quad 1 \le j \le m. \tag{12}$$

Note that we have $\theta_m = L(K - 1) - n = -|B(K)|$. The value θ_j has a rather intuitive interpretation. Indeed, regardless of the number of stable elements at the beginning of step j, θ_j gives the difference between the number of non-stable elements to pop and the number of new stable elements at the end of step j. Clearly, it is worth performing a reordering at step BI_j if θ_j is negative.

Theorem 3. *We obtain an optimal strategy for an elementary loading sequence by reordering at each step BI_j such that $\theta_j < 0$; the minimum number of pop operations is:*

$$L^* = n + \sum_{j=1}^{m} \min\{0, \theta_j\}. \tag{13}$$

Proof. The thesis follows from the fact that a strategy reordering at steps in the set $I \subseteq BI$ takes $L^{(I)}$ pops, where

$$L^{(I)} = n + \sum_{BI_j \in I} \theta_j.$$

This can be proved by induction on the length k of I. For $k = 1$ we necessarily have $I = \{BI_m\}$, i.e., the simple strategy that reorders the whole stack at step K, taking $L(K-1) = n - |B(K)| = n + \theta_m$ pops. The induction step follows immediately from the definition of the values θ. □

Example 2 (*Continued*). Consider the subsequences $B^{(1)}$ and $B^{(2)}$ in Fig. 6. Recall that we have $BI^{(1)} = \{3, 4\}$ and $ST^{(1)} = \{1, 7\}$. We obtain the values $\theta_1^{(1)} = L^{(1)}(BI_1^{(1)} - 1) - ST_1^{(1)} = 4 - 1 = 3$ and $\theta_2^{(1)} = L^{(1)}(BI_2^{(1)} - 1) - ST_2^{(1)} = 5 - 7 = -2$. From (13) we obtain $L^* = n_2^{(1)} + \theta_2^{(1)} = 7 - 2 = 5$, corresponding to the delayed reordering at step 4.

The non-increasing sequence for $B^{(2)}$ is $BI^{(2)} = \{7, 8\}$, and we have $ST^{(2)} = \{2, 6\}$. We obtain the values $\theta_1^{(2)} = L^{(2)}(BI_1^{(2)} - 1) - ST_1^{(2)} = 1 - 2 = -1$ and $\theta_2^{(2)} = L^{(2)}(BI_2^{(2)} - 1) - ST_2^{(2)} = 3 - 6 = -3$. Here we reorder at steps 7 and 8, and from (13) we obtain $L^* = n_2^{(2)} + \theta_1^{(2)} + \theta_2^{(2)} = 6 - 1 - 3 = 2$. Note that the delayed reordering at step 8 requires 3 pops. □

So far, we have shown how to decompose a loading sequence into elementary subsequences, and how to find an optimal strategy for an elementary loading sequence. Clearly, this allows us to deal with arbitrary loading sequences, by applying decomposition if necessary. In the following we show that the solution approach can be extended to non-elementary sequences.

Suppose that the loading sequence B is decomposed into q elementary subsequences $B^{(l)} = \{B(p^{(l)}), \ldots, B(r^{(l)})\}$, $l = 1, \ldots, q$. Let $n^{(l)}$ be the length of $B^{(l)}$, that is, the sum of the cardinalities of the batches in $B^{(l)}$. Let $BI^{(l)} = \{BI_{i^{(l)}}, \ldots, BI_{m^{(l)}}\}$ denote the subsequence of BI corresponding to $B^{(l)}$, and let

$$R = \{BI_1, \ldots, BI_m\} \setminus \bigcup_{l=1}^{q} BI^{(l)}$$

the (possibly empty) set of batches in BI that do not belong to any subsequence. As discussed above, we have $BI_j \in R$ if and only if BI_j can be ignored or simplified. According to (10), in this situation we either ignore or drop k elements, where:

$$k = ST_j - L(BI_{j-1}) = -\theta_j.$$

We therefore obtain:

$$n = \sum_{l=1}^{q} n^{(l)} - \sum_{BI_j \in R} \theta_j.$$

Recall that a decomposable strategy deals with each elementary sequence separately. As long as we have $i, j \in BI^{(l)}$, $\rho(i, j)$ correctly gives the number of pops required to reorder at step j after step i. We conclude that each subsequence $B^{(l)}$ requires $L^{(l)}$ pop operations, where

$$L^{(l)} = n^{(l)} + \sum_{j=i^{(l)}}^{m^{(l)}} \min\{\theta_j, 0\}$$

is obtained from (13). The total number of pop operations is given by:

$$\begin{aligned}
\sum_{l=1}^{q} L^{(l)} &= \sum_{l=1}^{q} \left(n^{(l)} + \sum_{j=i^{(l)}}^{m^{(l)}} \min\{\theta_j, 0\} \right) \\
&= \sum_{l=1}^{q} n^{(l)} + \sum_{l=1}^{q} \sum_{j=i^{(l)}}^{m^{(l)}} \min\{\theta_j, 0\} \\
&= n + \sum_{BI_j \in R} \theta_j + \sum_{l=1}^{q} \sum_{j=i^{(l)}}^{m^{(l)}} \min\{\theta_j, 0\} \\
&= n + \sum_{j=1}^{m} \min\{\theta_j, 0\}.
\end{aligned}$$

Let us evaluate the computational complexity of our solution method. We assume that each batch in B is represented by a set (e.g. a list) of elements, with corresponding types. It is easy to see that high and L can be built in $O(n)$ time given B; as shown before, BI and NS can be built in $O(K)$ time given high. Furthermore, θ can be computed in $O(K)$ time once the values ST are given. As shown in the Appendix, the values ST can be computed in $O(n)$ time. Thus we can state the following theorem.

Theorem 4. *The stack loading problem can be solved in O(n) time. We obtain an optimal loading strategy by reordering at each step BI_j that cannot be simplified or ignored and such that $\theta_j < 0$; the minimum number of pop operations is provided by (13).*

4. The stack loading–unloading problem

Now let us consider the case where we have a loading phase, where K batches are loaded into the stack, followed by an unloading phase where the elements are removed from the stack according to their type. We assume that the stack has not to be necessarily ordered at the end of the loading phase, thus we adopt a *mixed* strategy where reorderings are allowed both in the loading and in the unloading phase. The objective is to minimize the sum of pop and push operations. A mixed strategy may give rise to a smaller number of operations with respect to the optimal "pure loading" and "pure unloading" strategies, that are particular cases of the mixed strategy.

Example 3. Consider the loading sequence $B(1) = \{3, 4\}, B(2) = \{4, 4, 5\}, B(3) = \{2, 3, 3\}, B(4) = \{1, 3, 5\}, B(5) = \{2, 3\}$. The optimal loading strategy requires ten pop operations. If no reordering is performed in the loading phase, the stack [2, 3, 1, 3, 5, 2, 3, 3, 4, 4, 5, 3, 4] is obtained, and ten push operations are required in the unloading phase. However, an optimal mixed strategy requires only 7 operations. The detailed solutions are reported in the Appendix.

Let us point out some basic properties of a mixed strategy. Clearly, in an optimal strategy, the unloading phase is optimal, thus the number of push operations is determined by the increasing sequence IS obtained at the end of the loading phase, in particular by the values δ. In the loading phase, we are interested in reordering operations that have a positive effect in the unloading phase, that is, aiming at the following goals:

- decrease deep(t) for a type t in IS;
- expand IS by including further types.

Furthermore, an optimal loading strategy must be dominating. If a reordering of depth h is performed at step $s > 1$, the substack $S[h, n]$ will not be further modified in the loading phase, i.e. it is "accepted", even if it is not necessarily sorted.

Observe that, according to the above properties, in the loading phase of a mixed strategy we build IS in reverse order (from IS_m to IS_1) as the "accepted" part of the stack grows. As we shall see more formally later, the whole loading process can be described by keeping track of the steps where a reordering is performed, and of the minimum type already inserted in IS at each step. Based on this observation, we shall devise a Dynamic Programming algorithm that implicitly enumerates all the mixed strategies. To this aim, we need to introduce further notations and properties.

4.1. Notations and basic properties

Throughout this section, the notations $IS = \{IS_1, \ldots, IS_m\}$ and deep(t) refer to the increasing sequence and deepest positions in the unloading phase; clearly, IS and deep are determined by the loading phase of the strategy.

Let S^C denote the stack configuration at the beginning of a generic loading step $s > 1$; clearly, S^C is a partially loaded stack, that is, a substack $S[h_s, n]$ where $h_s = n - L(s-1) + 1$. By $\phi^C(t)$ we denote the deepest position of type t in S^C, where we assume $\phi^C(t) = 0$ if no elements of type t have been loaded yet. Let $IS^C = \{IS^C_1, \ldots, IS^C_q\}$ denote the *partial* increasing sequence for S^C, defined by those types t with $\phi^C(t) > 0$ satisfying Condition (2); clearly, we have $\phi^C(IS^C_q) = n$, while IS^C_1 is the smallest type appearing in the first $s - 1$ steps.

As discussed above, any useful reordering at step s should aim at modifying IS^C and/or ϕ^C; an important property follows.

Property 3. *In the loading phase of an optimal mixed strategy a reordering has depth $h = \phi^C(t)$ for some type $t = IS^C_j$ in the current partial increasing sequence IS^C; moreover, we have $S[h] > t$ after the reordering.*

Proof. To prove the first claim, observe that a reordering of depth $h < \phi^C(IS^C_1)$ has no effect on IS^C and ϕ^C, while if $\phi^C(IS^C_i) < h < \phi^C(IS^C_{i+1})$ a reordering has the same effect as if $h = \phi^C(IS^C_i)$. For what concerns the second claim: if $S[h] = t$ then the same effect is obtained by a reordering at depth $h' = \phi^C(IS^C_{j-1})$, if $j > 1$, or by no reordering at all if $j = 1$. □

Let us consider a reordering of depth $h = \phi^C(IS^C_j)$: the choice of h can be interpreted as follows. First of all, the final part of IS^C, say $IS^F = \{IS^C_{j+1}, \ldots, IS^C_q\}$, is fixed as the final part of IS, that is, we shall have $IS_{i+m-q} = IS^C_i$ for $j < i \leq q$. We refer to IS^F as the *frozen (increasing) sequence*, and we say that the types in IS^F are *frozen*. Moreover, type IS^C_j is not frozen yet, indeed, the reordering decreases deep(IS^C_j), which may be further decreased at later steps.

We have thus established an important principle: the target of a reordering at step s is the *maximum, not yet frozen* type in IS^C. Now let us say that the *minimum frozen type* is IS^C_{j+1} if the frozen sequence $IS^F = \{IS^C_{j+1}, \ldots, IS^C_q\}$ is not empty, and $T + 1$ otherwise. We can refine Property 3 as follows.

Property 4. *The depth h of a reordering at step s, if any, is uniquely determined by the current minimum frozen type. Formally, $h = \phi^C(IS^C_j)$, where IS^C_j is the largest type in IS^C smaller than the current minimum frozen type.*

Definition 3. A mixed strategy is *regular* if the loading phase fulfills Property 4.

In a regular strategy, we decide at each step if and how to expand the frozen sequence, and this determines the depth of the (possible) reordering; the loading part of the mixed strategy ends as soon as IS cannot be further extended, that is, when type 1 is frozen. Note that no reordering can be performed at step s if all the types in IS^C are frozen. In particular, we obtain a "pure unloading" regular strategy if we choose to freeze each possible type (that is, each type in IS^C) at each step.

Observe that a regular strategy is not necessarily dominating. If we have $S[h] < IS^C_{j+1}$ after a reordering at depth $h = \phi^C(IS^C_j)$, a second reordering with the same minimum frozen type IS^C_{j+1} has depth h too; actually, the first reordering is pointless here. Clearly, an optimal mixed strategy must be regular *and* dominating.

Example 3 (*Continued*). Suppose that we perform the first reordering at step $s = 3$; we have $S^C = S[9, 13] = [4, 4, 5, 3, 4]$ and $IS^C = [3, 4]$. Assuming minimum frozen type $IS^C_2 = 4$, the depth is $\phi^C(3) = 12$, and we obtain the new partial stack $S^C = S[6, 13] = [2, 3, 3, 3, 4, 4, 5, 4]$, with $IS^C = [2, 3, 4]$; we may thus freeze type 3 and type 2 as well.

Now consider the situation at step $s = 4$. If the minimum frozen type is 4 or 3 then we can perform a reordering of depth $9 = \phi^C(3)$ or $6 = \phi^C(2)$, respectively. If the minimum frozen type is 2, no reordering can take place at step 4; in this case, a reordering of depth $\phi^C(1) = 3$ is possible at step $s = 5$, when $S^C = S[3, 13] = [1, 3, 5, 2, 3, 3, 3, 4, 4, 5, 4]$ and $IS^C = [1, 2, 3, 4]$. □

We shall now provide a confluence property stating that, for a given step and minimum frozen type, the result of a reordering is (in some sense) unique. Recall that $C(s, t)$ denotes the number of type t elements in batch $B(s)$. We denote by $PL(s, t) = \sum_{s'=1}^{s} C(s', t)$ the number of elements of type t loaded in the first s steps, and by $CL(s, t) = \sum_{t'=t}^{T} PL(s, t')$ the total number of elements of type greater than or equal to t loaded in the first s steps. We let $PL(s, T + 1) = CL(s, T + 1) = 0$ for each $1 \le s \le K$. Note that $L(s) = \sum_{t=1}^{T} PL(s, t) = CL(s, 1)$.

Definition 4. A (partially loaded) stack is (s, t)-ordered if it contains the first s batches, and all the elements of type smaller than t, if any, appear ordered on top of the stack. In particular, an ordered stack containing the first s batches is $(s, T + 1)$-ordered.

In an (s, t)-ordered stack the elements of a type $t' < t$ such that $PL(s, t') > 0$ appear in the substack $S[h, k]$, where $k = n - CL(s, t' + 1)$ and $h = n - CL(s, t') + 1$. Note that h and k are determined by the problem data, and do not depend on the loading strategy. Clearly, an (s, t)-ordered stack is (s, t')-ordered for each $t' < t$. It is easy to see that if S^C is an $(s - t)$-ordered stack then each type $t' < t$ with $PL(s, t') > 0$ belongs to IS^C.

Property 5. *A reordering at step s with minimum frozen type t yields an (s, t)-ordered stack.*

Proof. According to Property 4, the reordering has depth $h = IS^C_j$, and there are no elements of type less than t in positions $h + 1, \ldots, n$. The claim than follows immediately from the definition of reordering. □

Theorem 5. *Suppose that at the end of step s the minimum frozen type is t and the stack is (s, t)-ordered; let $IS^F = \{IS_j, \ldots, IS_m\}$ denote the current (possibly empty) frozen sequence. Then, in a regular strategy:*

1. *the reorderings in steps $s + 1, \ldots, K$ do not affect IS^F and $\mathrm{deep}(IS_i)$ for $j \le i \le m$;*
2. *the initial part $IS^I = \{IS_1, \ldots, IS_{j-1}\}$ of the increasing sequence does not depend on the reorderings performed in steps $2, \ldots, s$.*

Proof. The first claim follows immediately from the definition of regular strategy. In order to prove the second claim, recall that IS^I contains only types smaller than t, and that the current stack is (s, t)-ordered. Therefore, the order of the elements in the substack $S[n - PL(t) + 1, n]$, that have type t or greater, does not affect IS^I. Moreover, the current configuration of the elements with type less than t, if any, does not depend on the reorderings in steps $2, \ldots, s$. □

In light of Theorem 5 the loading phase of a regular strategy can be described by a sequence of step-type pairs $\{(s_1, t_1), \ldots, (s_k, t_k)\}$, where $s_i < s_{i+1}$ and $t_i \le t_{i+1}$ for each $1 \le i < k$, meaning that at each step s_i we perform a reordering with minimum frozen type t_i. An empty sequence represents the "pure unloading" strategy. In fact, Theorem 5 states that the meaning of a reordering (s_{i+1}, t_{i+1}) is completely determined by the pair (s_i, t_i), and does not depend on the whole subsequence $\{(s_1, t_1), \ldots, (s_i, t_i)\}$.

4.2. The loading–unloading state graph

We can now define the *state graph* $G = (N, A)$ used in our Dynamic Programming algorithm. A state, that is a node in G, is a pair (s, t) representing the situation where the current stack is (s, t)-ordered and t is the minimum frozen type. There is a node $(s, t) \in G$ for each $1 \le s \le K$ and t such that $PL(s, t) > 0$. Furthermore, for each $1 \le s \le K$ there is a node $(s, T + 1)$ representing the situation where the partial stack is completely ordered at the end of step s, and no type is frozen yet. Here

we assume that nodes are arranged in a grid, where rows (columns) correspond to types (steps); moreover, we assume that $(1, T+1)$ is the top-right corner of the grid.

Let us consider the state transitions corresponding to directed arcs between nodes in G. We distinguish two types of transition: from a node (s, t) to a node (s, t'), corresponding to *vertical* arcs, and from a node (s, t) to a node (s', t), corresponding to *horizontal* arcs.

The meaning of a vertical arc is that we extend the current frozen sequence by freezing one more type. In order to define these arcs formally, we need the following notation. Let $Tmin(s) = \min\{t : PL(s, t) > 0\}$ be the minimum type appearing in the first s batches. Given a state (s, t) in G such that $Tmin(s) < t$, let $nt(s, t) = \max\{t' < t : PL(s, t') > 0\}$ be the maximum type smaller than t contained in the first s batches. We let $nt(s, t) = 0$ for $t = Tmin(s)$. A vertical arc links each node (s, t) such that $Tmin(s) < t$ to the node (s, t'), where $t' = nt(s, t)$. No vertical arcs leave a node $(s, Tmin(s))$. Recall that an (s, t)-ordered stack is (s, t')-ordered for each $t' < t$, thus the transition to a state $(s, nt(s, t))$ is correct.

A horizontal arc links a node (s, t), where $s < K$, to a node (s', t), where $s' > s$. We distinguish two types of horizontal arcs, depending on whether the corresponding transition requires a reordering or not. If the transition from state (s, t) to state $(s+1, t)$ requires no reordering we add a *no-reordering arc* from (s, t) to $(s+1, t)$, which is the only arc leaving node (s, t). Otherwise, we add a *reorder arc* from (s, t) to each state (s', t) with $s < s' \leq K$.

To complete the state graph, we introduce a *target* state $(K, 0)$, representing the completion of the unloading phase; this node is connected by a vertical arc to node $(K, 1)$, which represents the end of the loading phase.

Remark 1. The transition from (s, t) to $(s+1, t)$ requires no reordering if, given an (s, t)-ordered stack, we obtain an $(s+1, t)$-ordered stack by pushing the elements in $B(s+1)$ in non-increasing order. This happens in exactly two cases:

(a) in state (s, t) all the type are frozen, that is, $t = Tmin(s, t)$;
(b) $\text{high}(s+1) \leq Tmin(s, t)$.

Note that the two cases are not mutually exclusive, and that for $t = 1$ case (a) holds, that is, we only have no-reordering arcs.

We assign costs to arcs as follows. Vertical arcs account for negative values δ in the unloading phase. A transition from (s, t) to (s, t'), where $1 < t \leq T$ and $t' = nt(s, t)$, implies that we shall have $IS_{l-1} = t'$ and $IS_l = t$ for some $1 < l \leq m$. Since we have an (s, t)-ordered stack when t' is frozen, we can compute the value

$$\text{deep}(t') = (n - CL(s, t'+1)) = (n - CL(s, t));$$

note that this value is determined by the problem data, and does not depend on the actual reorderings. Thus we can compute the value

$$\delta_l = \text{deep}(t') - U(t) = n - CL(s, t) - U(t) \tag{14}$$

and we set the cost of the vertical arc to $\min\{0, \delta_l\}$. Note that the vertical arc leaving node $(s, T+1)$, for $1 \leq s \leq K$, does not correspond to a value δ, and has a zero cost; in fact, the "effect" of this arc is to set $IS_m = nt(s, T+1)$. Finally, the arc from $(K, 1)$ to the target state $(K, 0)$ has a cost $= n - U(1)$, that is, the number of push operations for a complete reordering at stage 1. Note that this accounts for the value $\delta_1 = -U(1)$.

The cost of a horizontal arc from (s, t) to (s', t), $s' > s$, is the number of pop operations required by the transition, and is obviously zero for a no-reordering arc. Otherwise, the cost is given by $h-(n-L(s'-1))$, where h is the depth of the reordering at step s'. In order to determine h, observe that we have $t > Tmin(s, t)$, since there exists a reordering arc leaving node (s, t); let $t' = nt(s, t)$. By Property 4, h is the deepest position containing an element of type t', thus we have $h = n - CL(s, t)$. This gives the cost:

$$L(s' - 1) - CL(s, t), \tag{15}$$

that again does not depend on the reorderings performed before step s'. Note that for $t = T + 1$ we have $CL(s, t) = 0$ and the cost is $L(s' - 1)$, that is, the size of the stack; indeed, the depth of the reordering is n in this case.

Definition 5. Each path P from $(1, T+1)$ to $(K, 0)$ in G defines a unique regular strategy μ_P, where the loading phase is described by the sequence of step-type pairs $\{(s_1, t_1), \ldots, (s_k, t_k)\}$ corresponding to nodes in P whose predecessor arc is a horizontal reordering arc.

Lemma 2. *Given a path P from $(1, T+1)$ to $(K, 0)$ in G, the corresponding mixed strategy μ_P is such that:*

1. *the total cost of horizontal arcs give the number of pop operations in the unloading phase;*
2. *the length m of IS is given by the number of vertical arcs minus one;*
3. *the total cost of vertical arcs in P gives the number of push operations in the unloading phase.*

Proof. It follows immediately from the definition of G and regular strategy, taking into account Property 5 and Theorem 5. □

Definition 6. Let $P^A(s,t)$ denote the "no-reordering" path from (s,t) to $(K,0)$ obtained as follows: at each node, follow the leaving vertical arc if it exists, otherwise, follow the (unique) no-reordering horizontal arc (that exists, see Remark 1).

In general, $P^A(s,t)$ is not the unique path from (s,t) to $(K,0)$ containing only vertical and no-reordering arcs. We denote by $P^U = P^A(1,T+1)$ the no-reordering path corresponding to the "pure unloading" strategy; note that P^U marks the south-east border of the grid graph.

Lemma 3. *Given a regular strategy μ described by the sequence of step-type pairs $\{(s_1,t_1),\ldots,(s_k,t_k)\}$ there exists a path $P = P_\mu$ in G that defines the strategy $\mu_P = \mu$.*

Proof. Assume the sequence is not empty, otherwise $P_\mu = P^U$. The proof is constructive. For the sake of simplicity, let $(s_0,t_0) = (1,T+1)$; for $1 \le i \le k$ we define the subpath from (s_{i-1},t_{i-1}) to (s_i,t_i) as follows. If necessary (i.e. $t_{i-1} > t_i$) follow the path $P^A(s_{i-1},t_{i-1})$ until a node (s,t_i) is reached; note that this is possible, since type t_i belongs to the frozen sequence at step s_i. Then follow no-reordering arcs as long as possible, finally, follow the reordering arc to node (s_i,t_i). Clearly, the final part of P_μ is $P^A(s_k,t_k)$. □

Theorem 6. *The minimum cost path P from $(1,T+1)$ to $(K,0)$ in G defines an optimal mixed strategy μ_P.*

Proof. It follows from Lemmas 2 and 3. □

4.3. Computational complexity

In light of Theorem 6, we can find an optimal mixed strategy by solving a shortest path problem on the state graph G. This can be done in $O(K^2T)$ time, since G is acyclic and contains $O(KT)$ nodes and $O(K^2T)$ arcs. Since (as will become clear later) the time required to build G and compute the costs of the arcs is $O(n+K^2T)$, the loading–unloading problem can be solved in $O(n+K^2T)$ time. However, a complexity $O(n+KT)$ can be achieved if we avoid the explicit generation of the state graph, and solve the shortest path problem on G without considering all the arcs. Our approach is briefly described below.

The main idea is to process the nodes in G in decreasing order of type, in $T+1$ different phases: in phase t we process all the nodes corresponding to the same type t. Thus in each phase we only consider horizontal arcs; at the end of the phase, the optimal distances are "propagated" downwards by means of vertical arcs. Note that we have at most $KT+1$ vertical arcs.

Next we show that each phase can be performed in time $O(K)$ by exploiting the structure of the costs. We consider a generic phase t, and denote by $c(s,s',t)$ the cost of the arc from (s,t) to (s',t). Observe that there are at most K no-reordering arcs to process in each phase, so we concentrate on reordering arcs. Let us say that (s,t) is a *reordering node* if there exist reordering arcs leaving it. Consider two reordering nodes (s_1,t) and (s_2,t), $s_2 > s_1$, and let $s = s_2+1$. For each $s' > s$ the following relation holds:

$$c(s_1,s',t) - c(s_1,s,t) = c(s_2,s',t) - c(s_2,s,t) = L(s'-1) - L(s-1) = L(s'-1) - L(s_2) > 0. \tag{16}$$

Now suppose that we already know the minimum distances d_1 and d_2 for nodes (s_1,t) and (s_2,t), respectively, and let

$$\Delta = d_1 + c(s_1,s,t) - (d_2 + c(s_2,s,t)).$$

As follows from (16), if $\Delta < 0$ then (s_1,t) is a better predecessor than (s_2,t) not only for node (s,t) but also for each node (s',t) with $s' > s$. In this situation, we do not need to consider (s_2,t) any more. If otherwise $\Delta > 0$ then node (s_2,t) is preferable, and we do not need to consider (s_1,t) any more. If $\Delta = 0$, either one of the two nodes can be ignored.

The above observations suggest an iterative process where a reordering node is dropped at each iteration; the phase terminates after at most $K-2$ iterations. If the nodes are processed in increasing order of step, each iteration (including the computation of d_1 and d_2) requires constant time, in fact, no more than five arcs are considered. A detailed description of a procedure performing a generic phase is given in the Appendix.

Let us now consider the time spent to build the data structures that allow us to represent G implicitly and to compute the costs of the processed arcs. In order to represent the graph G, we need two sets of information: the function $nt(s,t)$, that defines the set of nodes and vertical arcs; and the function $Tmin(s)$, that together with $\mathrm{high}(s)$ allow us to identify no-reordering arcs, and thus horizontal arcs. In order to compute the arc costs accordingly to (14) and (15) we need the function $CL(s,t)$; here we assume that the functions $L(s)$ and $U(t)$ are given.

Functions $nt(s,t)$ and $CL(s,t)$ can be represented by means of two-dimensional, $K \times T$ arrays; $Tmin(s)$ only requires a liner array. Moreover, these functions can be easily computed in $O(KT)$ time once the function $PL(s,t)$ is known. In turn, function $PL(s,t)$ can be represented by a $K \times T$ array, and can be computed in $O(n+KT)$ time given a suitable representation of the batches, such as the one used in the Appendix to compute the values ST for the loading problem. The computational details are rather straightforward, and are omitted here. When the above array representation is used, each arc considered by the algorithm requires a constant processing time. Our last result follows.

Theorem 7. *The loading–unloading problem can be solved in $O(n+KT)$ time.*

Note that our algorithm is optimal for the instances where $n = \Omega(KT)$. Furthermore, we obtain an $O(KT)$ complexity if the input is encoded as in [1], that is, by means of a $K \times T$ *shipment matrix* A, where $A_{st} = C(s,t)$ is the number of elements of type t in batch $B(s)$. Indeed, the array representing $PL(s,t)$ can be easily computed from A in $O(KT)$ time. Note that our algorithm is optimal with respect to this input encoding.

5. Conclusions

In this paper we considered a class of combinatorial optimization problems related to piling objects in stacks. We defined three problems in this class, and for each of them we proposed a polynomial time algorithm which improves the known algorithms. Furthermore, for two of the three problems we provided a closed form solution, which allows us to compute the cost of each "reasonable" sub-optimal solution. As far as we know, there is no closed form solution in the literature for this class of problems.

These problems derive as an abstraction from many practical problems arising in applicative contexts, as for example in logistics and transportation. Many authors have considered in great detail similar problems, arising in fields such as freight rail composition, container stowage or vehicle routing, where the constraints imposed by the specific application prevailed on the general combinatorial structure.

We believe that our contribution provides an abstract and simple combinatorial framework which leaves the opportunity for further investigations, both from a theoretical and a practical point of view. Indeed, slight extensions of the problems defined here may provide useful tools for approaching real applications. Moreover, the proposed framework could be adopted to better understand complexity issues related to other problems, such as the two-stacks and three-stacks reshuffling problems [4], whose computational complexity is still open.

Acknowledgments

The authors are grateful to the anonymous referees for their comments and for helpful references.

Appendix

A.1. Computing the number of stable elements

Here we show that ST can be built in $O(n)$ time once BI and NS are given. In order to compute ST in $O(n)$ time, we adopt an *incremental* approach; in particular, we obtain each ST_i from ST_{i-1} by adding the number of elements that can become stable at step BI_i. To this aim, we need to introduce an auxiliary counter array $LT[t]$, $t = 1, \ldots, T$, whose elements are initially set to zero. At the end of each step s, $LT[t] = \sum_{j=1}^{s} C(s,t)$ gives the number of elements of type t loaded in the stack at the end of step s. More precisely, during step s we perform an updating $LT[t] := LT[t] + 1$ for each element of type t in the batch $B(s)$; clearly, we can maintain LT with an overall $O(n)$ cost.

The algorithm maintains a counter $Stot$, that gives the current number of stable elements, and the *minimum stable type* $T_{\min}$, such that stable elements have type at least $T_{\min}$. At each step BI_i, $i < m$, the threshold $T_{\min}$ is updated by setting $T_{\min} = \min\{T_{\min}, NS_{i+1}\}$; here we exploit the fact that an element of type at least NS_{i+1} can become stable at step BI_i. Each time the threshold $T_{\min}$ decreases, the current value $LT[t]$ is added to $Stot$ for each type t that becomes stable. Initially, $Stot = 0$ and $T_{\min} = NS_2$; here we assume $m > 1$, otherwise the loading problem is trivial.

The algorithm computing the values ST is described below. It is easy to see that the whole process takes $O(n)$ time.

Step 0 set $ST_0 = 0, s = 1, i = 1, T_{\min} = NS_2, Stot = 0$;
Step 1 for each $e \in B(s)$:
- let t be the type of e;
- set $LT[t] = LT[t] + 1$;
- if $t \geq T_{\min}$ set $Stot = Stot + 1$;

Step 2 set $s = s + 1$; if $s \leq BI_i$ go to Step 1; otherwise, go to Step 3;
Step 3 set $ST_i = Stot$; $i = i + 1$; if $i = m$ go to Step 5; otherwise, go to Step 4;
Step 4 if $T_{\min} > NS_{i+1}$: for each $T_{\min} > t \geq NS_{i+1}$ set $Stot = Stot + LT[t]$; set $T_{\min} = NS_{i+1}$ and go to Step 1;
Step 5 set $ST_m = n$ and STOP.

A.2. Loading–unloading problem: An example

We have the loading sequence $B(1) = \{3, 4\}$, $B(2) = \{4, 4, 5\}$, $B(3) = \{2, 3, 3\}$, $B(4) = \{1, 3, 5\}$, $B(5) = \{2, 3\}$; thus $n = 13$ and $K = T = 5$. The non-increasing sequence is $NS = \{2, 4, 5\}$, and we have $\theta_1 = 2 - 1 = 1$, $\theta_2 = 8 - 9 = -1$, $\theta_3 = 11 - 13 = -2$. Thus, the optimal loading strategy needs $13 - 1 - 2 = 10$ pop operations, 8 in step $NS_2 = 4$ and 2 in step $NS_3 = 5$.

If no reordering is performed in the loading phase, we obtain the stack

$$[2, 3, 1, 3, 5, 2, 3, 3, 4, 4, 5, 3, 4].$$

The increasing sequence is $IS = \{1, 2, 3, 4\}$, with $\delta_1 = -U(1) = -1$, $\delta_2 = \text{deep}(1) - U(2) = 3 - 3 = 0$, $\delta_3 = \text{deep}(2) - U(3) = 6 - 8 = -2$ and $\delta_4 = \text{deep}(3) - U(4) = 12 - 11 = 1$. The optimal unloading strategy performs reorderings at stages 1 and 3; the total number of push operations is $n + \delta_1 + \delta_3 = 10$. There is an anticipated reordering of types 2 and 4 at stages 1 and 3, respectively; in both stages, the number of push operations is 5.

The loading–unloading problem admits two optimal mixed strategies with an overall number of operations equal to 7. The loading phase of the first one is as follows:

- in step 2, a reordering of depth n requires two pops and gives the substack $S[9, 13] = [3, 4, 4, 4, 5]$;
- in step 3, the substack becomes $S[6, 13] = [2, 3, 3, 3, 4, 4, 4, 5]$ (no reordering);
- in step 4: we have $IS^C = [2, 3, 4, 5]$; a reordering of depth $\phi^C(2) = 6$ (i.e., with minimum frozen type 3) requires one pop and gives the substack $S[3, 13] = [1, 2, 3, 5, 3, 3, 3, 4, 4, 4, 5]$;
- in step 5: we have $IS^C = [1, 2, 3, 4, 5]$; a reordering of depth $\phi^C(1) = 3$ (i.e., with minimum frozen type 2) requires one pop and gives the stack $S = [1, 2, 3, 2, 3, 5, 3, 3, 3, 4, 4, 4, 5]$.

The total number of pop operations is 4. In the unloading phase, we have $IS = [1, 2, 3, 4, 5]$, with

$$\delta_2 = \text{deep}(1) - U(2) = 1 - 3 = -2$$
$$\delta_3 = \text{deep}(2) - U(3) = 4 - 8 = -4$$
$$\delta_4 = \text{deep}(3) - U(4) = 9 - 11 = -2$$
$$\delta_5 = \text{deep}(4) - U(5) = 12 - 13 = -1.$$

The optimal unloading strategy performs reorderings at each stage (i.e., no anticipated reordering); the total number of push operations is 3.

The second optimal mixed strategy is similar, except for the reordering at step 2; here the frozen type is 4 and we reorder at depth $12 = \phi^C(3)$, which requires a single pop. Thus the loading phase requires 3 pops, and yields the stack

$$S = [1, 2, 3, 2, 3, 5, 3, 3, 3, 4, 4, 5, 4];$$

Now we have $IS = [1, 2, 3, 4]$, with the same values $\delta_1, \ldots, \delta_4$ as before. The optimal unloading strategy reorders at each stage in the increasing sequence, and requires 4 push operations.

The state graph G is given in Fig. 7. Recall that rows and columns correspond to types (actually, values $0, \ldots, T + 1$) and steps, respectively. Dotted arrows represent no-reorder horizontal arcs. Nodes corresponding to pure-loading strategies are filled. The two optimal strategies correspond to the two marked paths. More precisely, the marked paths are the ones obtained according to the construction described in the proof of Lemma 3. Note however that G contains other origin-destination paths with the same cost, containing nodes (3, 5) and/or (3, 4).

Note that the two paths join at node (2, 4), indeed, both strategies yield a (2, 4)-ordered stack, although the stack configurations at the end of step 2 are different: [3, 4, 4, 4, 5] for the first strategy and [3, 4, 4, 5, 4] for the second one.

Below we report the main functions, that is, linear and bidimensional arrays, used in our solution algorithm. First of all, $Tmin(s)$ and high(s) are as follows.

$s =$	1	2	3	4	5
$Tmin(s) =$	3	3	2	1	1
$high(s) =$	4	5	3	5	3

Here are the two matrices representing the functions PL and CL; rows and columns correspond to steps $1, \ldots, 5$ and types $1, \ldots, 5$, respectively. Note that the function $L(s)$ can be read in the first column of CL.

$PL =$

0	0	1	1	0
0	0	1	3	1
0	1	3	3	1
1	1	4	3	2
1	2	5	3	2

$CL =$

2	2	2	1	0
5	5	5	4	1
8	8	7	4	1
11	10	9	5	2
13	12	10	5	2

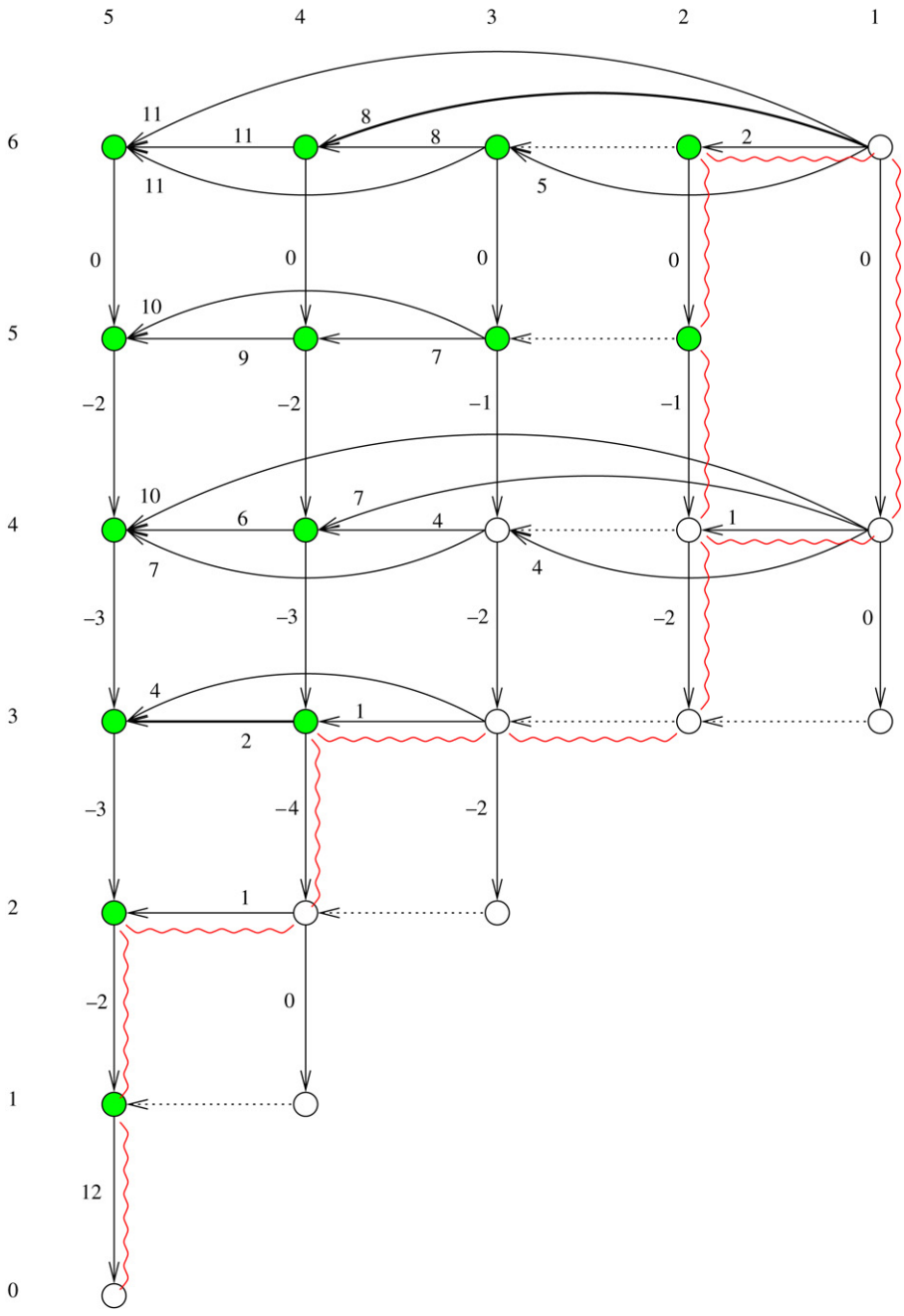

Fig. 7. The state graph G.

Finally, we provide the function $nt(s, t)$; in this case, rows and columns correspond to steps $1, \ldots, 5$ and types $1, \ldots, 6 = T + 1$, respectively. We let $nt(s, t) = \infty$ if (s, t) is not a node in G.

∞	∞	0	3	∞	4
∞	∞	0	3	4	5
∞	0	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	5

A.3. Computing shortest paths in the state graph

Here we give some details about the $O(KT)$ algorithm for computing a shortest path in G. Procedure Phase(t) finds the optimal distance $d(s,t)$ for each node (s,t) in G. The label $v(s,t)$ denotes the "vertical" distance value for (s,t), propagated downwards via vertical arcs at the end of previous phases. Initially, each $v(s,t)$ is set to $+\infty$; at the end of phase t, $v(s, nt(s,t))$ is updated for each node (s,t) with $nt(s,t) > 0$. The function $c(s,s',t)$ gives the cost of the reorder arc from (s,t) to (s',t).

The first node to process in phase t is found using function first(t), that returns the minimum step s such that $PL(s,t) > 0$; clearly, this function takes $O(K)$ time. The recursive function *next_rn* (s,t) (given below) finds the reorder node (s',t) with minimum $s' \geq s$, if it exists. It returns $s' = s$ if (s,t) is a reorder node, or if $s = K$; otherwise, it issues a recursive call to *next_rn* $(s+1,t)$. Assuming that $d(s,t)$ is the optimal distance for (s,t) when *next_rn* (s,t) is called, the function sets the optimal distance $d(s+1,t) = \min\{v(s+1,t),\ d(s,t)\}$ before the recursive call. It is easy to see that *next_rn* is called at most once for each node.

procedure Phase(t)

step 0 $s_1 := \text{first}(t)$; $d(s_1,t) := v(s_1,t)$; $s_1 := next\_rn(s_1,t)$;
if $s_1 = K$ then STOP; otherwise,
/* *label* $s_1 + 1$ */
$s_2 := s_1 + 1$; $d(s_2,t) := \min\{v(s_2,t),\ d(s_1,t) + c(s_1,s_2,t)\}$;
step 1 $s_2 := next\_rn(s_2,t)$; if $(s_2 = K)$ then STOP; otherwise, $s := s_2 + 1$;
step 2 let $d(s,t) := \min\{v(s,t),\ d(s_1,t) + c(s_1,s,t),\ d(s_2,t) + c(s_2,s,t)\}$;
let $\Delta := d(s_1,t) + c(s_1,s,t) - d(s_2,t) - c(s_2,s,t)$;
if $\Delta > 0$ then $s_1 := s_2$; /* *skip* s_1 */
step 3 $s_2 := s$; GOTO step 1;

function *next_rn*(s,t)
if $s = K$ **or** $(t > Tmin(s,t)$ **and** $\text{high}(s+1) > Tmin(s,t))$ **then return** (s);
$d(s+1,t) := \min\{v(s+1,t),\ d(s,t)\}$; **return** *next_rn*$(s+1,t)$;
end_procedure

The optimal labels $d(s,t)$ for our example are given in the next table; rows and columns correspond to steps $1,\ldots,5$ and types $0,1,\ldots,6 = T+1$, respectively. We let $d(s,t) = \infty$ if (s,t) is not a node in G.

∞	∞	∞	0	0	∞	0
∞	∞	∞	-1	1	2	2
∞	∞	-3	-1	1	2	2
∞	-4	-4	0	5	8	8
7	-5	-3	2	8	11	11

Let us describe the application of procedure Phase(t) for $t = 4$. At the beginning, we have the "vertical" labels $v(1,4) = 0$, $v(2,4) = v(3,4) = 1$, $v(4,4) = 6$ and $v(5,4) = 9$. In Step 0, we get $s_1 = \text{first}(4) = 1$ and we set $d(1,4) = v(1,4) = 0$; moreover, we set $d(2,4) := \min\{1, 0+1\} = 1$. In step 1, a call to *next_rn*$(2,4)$ returns $s_2 = 3$, after setting $d(3,4) := \min\{1, 1+0\} = 1$; we thus set $s = 4$. In step 2, given the arc costs $c(1,4,4) = 7$ and $c(3,4,4) = 4$ we set $d(4,4) = \min\{6, 0+7, 1+4\} = 5$ and $\Delta = 7 - 5 = 2$. Since $\Delta > 0$ we skip node $(1,4)$, that is, we set $s_1 = 3$. Then we jump back to step 1, were we set $s_2 = next\_rn(4,4) = 4$ and $s = 5$. In step 2, given the arc costs $c(3,5,4) = 7$ and $c(4,5,4) = 6$ we set $d(5,4) = \min\{9, 1+7, 5+6\} = 8$ and $\Delta = 8 - 11 = -3$. Since $\Delta < 0$ we do not change s_1, that is, we skip node $(4,4)$. When we jump back to step 1 we have $s_2 = 5$, and the procedure stops.

References

[1] A. Aslidis, Combinatorial algorithms for stacking problems, Ph.D. Thesis, MIT Boston, 1989.
[2] A. Aslidis, Minimization of overstowage in containership operations, in: H.E. Bradley (Ed.), Operational Research '90, Pergamon Press, 1991, pp. 457–471.
[3] M. Avriel, M. Penn, N. Shpirer, S. Witteboon, Stowage planning for container ships to reduce the number of shifts, Annals of Operations Research 76 (1998) 55–71.
[4] M. Avriel, M. Penn, N. Shpirer, Container ship stowage problem: Complexity and connection to the coloring of circle graphs, Discrete Applied Mathematics 103 (2000) 271–279.
[5] T. Crainic, J.-A. Ferland, J.-M. Rousseau, Multicommodity, multimode freight transportation: A general modeling and algorithmic framework for the service network design problem, Transportation Research B 20B (1986) 225–242.
[6] A. Imai, K. Sasaki, E. Nishimura, S. Papadimitriou, Multi-objective simultaneous stowage and load planning for a container ship with container rehandle in yard stacks, European Journal of Operational Research 171 (2006) 373–389.
[7] G. Levitin, R. Abezgaouz, Optimal routing of multiple-load AGV subject to LIFO loading constraints, Computers & Operations Research 30 (2003) 397–410.
[8] D. Steenken, S. Voss, R. Stahlbock, Container terminal operation and operations research — a classification and literature review, OR Spectrum 26 (1) (2004) 3–49.
[9] D. Steenken, T. Winter, U. Zimmermann, Stowage and transport optimization in ship planning, manuscript, University of Braunschweig, 2001. Available at citeseer.ist.psu.edu/steenken01stowage.html.
[10] H. Xu, Z. Chen, S. Rajagopal, S. Arunapuram, Solving a practical pickup and delivery problem, Transportation Science 37 (2003) 347–364.